Organization of Programming Languages
CS3200 / 5200N

Lecture 06

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Data Types

- A **data type** defines a collection of data objects and a set of predefined operations on those objects.
- **Primitive data types** are those not defined in terms of other data types:
  - Some primitive data types are merely reflections of the hardware.
  - Others require only a little non-hardware support for their implementation.
- **User-defined types** are created with flexible *structure* defining operators (ALGOL 68).
- **Abstract data types** separate the interface of a type (visible) from the representation of that type (hidden).

Primitive Data Types

- **Integers** – almost always an exact reflection of the hardware.
  - Java's signed integers: `byte`, `short`, `int`, `long`.
- **Floating Point** – model real numbers, but only as approximations.
  - Support for two types: `float` and `double`.
- **Complex** – two floats, the real and the imaginary.
  - Supported in Fortran and Python.
- **Boolean** – two elements, `true` and `false`.
  - Implemented as bits or bytes.
- **Character** – stored as numeric codings.
  - ASCII 8-bit encoding, UNICODE 16-bit encoding.

Primitive Data Types

- **Rationals**:
  - represented as pairs of integers (Scheme, Common LISP):
    - `(rational?  6/10)  =>  #t`
- **Decimals**:
  - use a base-10 encoding to avoid round-off in financial arithmetic.
    - Cobol, PL/I.

Scalar Types

- **Scalar types** (also **simple types**):
  - All primitive types.
  - Some user-defined types:
    - **Fixed-point**:
      - represented as integers, with position for decimal point:
        - » type Fixed_Point is delta 0.01 digits 10;
    - **Enumerations**:
      - represented as small integers:
        - » type weekday is (sun, mon, tue, wed, thu, fri, sat);
    - **Subranges**:
      - subtype workday is weekday range mon . . fri;

Composite Types

- **Records** (structures)
- **Variant records** (unions)
- **Arrays**
  - **Strings** (arrays of characters)
- **Sets**
- **Pointers**
- **Lists**
- **Files**

Array Types

- An **array** is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
- Indexing is a mapping from indices to elements:

  `array_name[index_value_list] → an element`

- Index range checking:
  - C, C++, Perl, and Fortran do not specify range checking.
  - Java, ML, C# specify range checking.
  - In Ada, the default is to require range checking, but it can be turned off.

Array Categories

- **Static**: subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency – no dynamic allocation/deallocation.
  - Example: arrays declared as `static` in C/C++ functions.

- **Fixed Stack-Dynamic**: subscript ranges are statically bound, but the allocation is done at declaration time (at run-time)
  - Advantage: space efficiency – stack space is reused.
  - Example: arrays declared in C/C++ functions without the `static` modifier.

Array Categories

- **Conformant Arrays**: array parameters where bounds are symbolic names rather than constants:
  - Pascal, Modula-2, Ada, C99.
    - C only supports single dimensional conformant arrays.

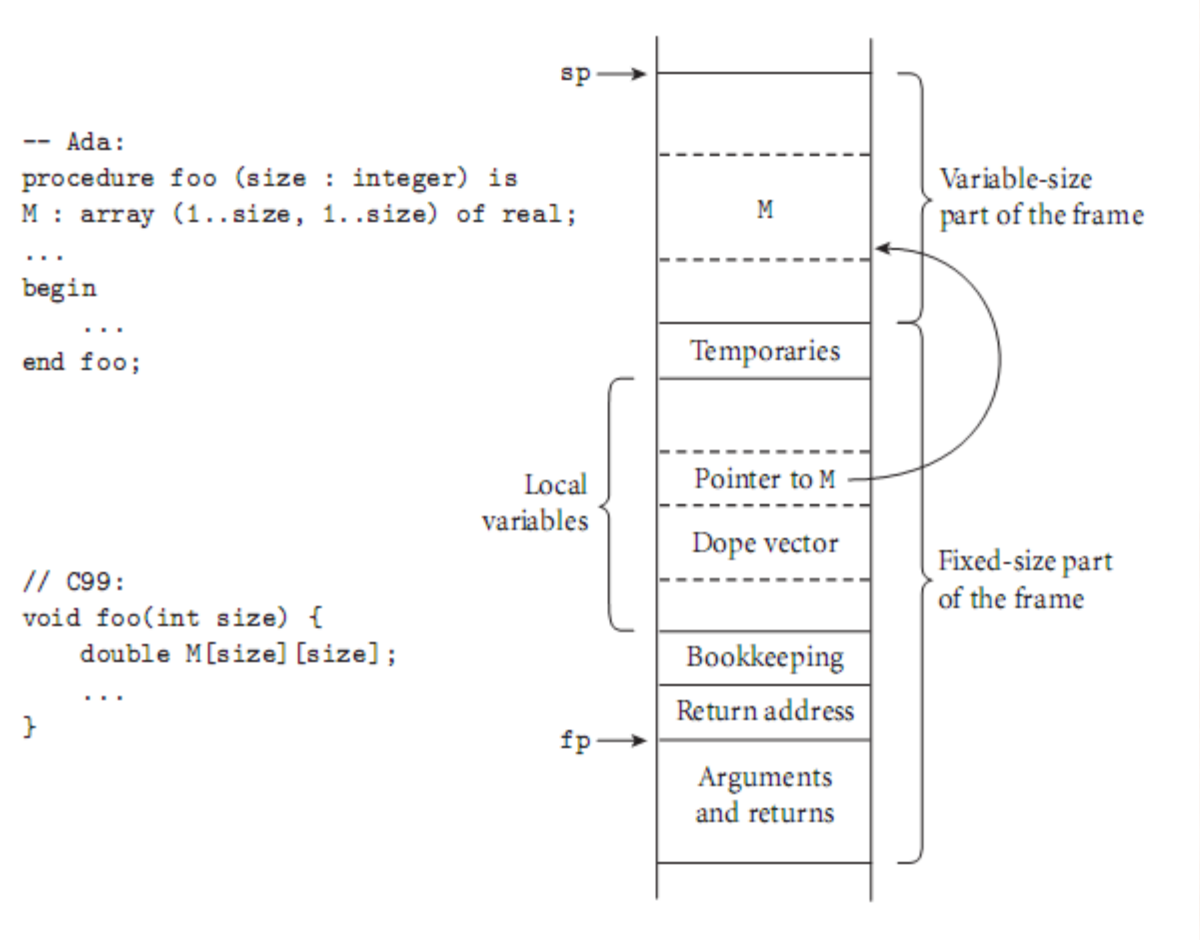
```
function DotProduct(A, B: array[lower .. uper : integer] of real) : real;

void square(int n, double M[n][n]);
```

Array Categories

- **Stack-Dynamic**: subscript ranges are dynamically bound and the storage allocation is dynamic (at run-time):
  - Advantage: flexibility – the size of an array need not be known until the array is to be used.
  - Example: Ada arrays, C99.

```
Get(List_Len);
declare
  List: array(1. . List_len) of Integer;
  begin
  …
  end;
```

Implementation of Stack Dynamic Arrays



```
-- Ada:
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
    ...
end foo;
```

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```

Array Categories

- **Fixed Heap-Dynamic**: similar to fixed stack-dynamic i.e. subscript range and storage binding are fixed after allocation:
  - Binding is done when requested by the program.
  - Storage is allocated from the heap.
  - Examples:
    - C/C++ using malloc/free or new/delete.
    - Fortran 95.
    - In Java all arrays are fixed heap-dynamic.
    - C#.

Array Categories

- **Heap-dynamic**: binding of subscript ranges and storage allocation is dynamic and can change any number of times:
  - Advantage: flexibility, as arrays can grow or shrink during program execution.
  - Examples:
    - C#:

```
ArrayList intList = new ArrayList();
intList.add(nextOne);
```

    - Java has a similar class, but no subscripting (use methods get()/set() instead).
    - Perl, JavaScript, Python, Ruby

Array Categories

- **Static shape** arrays:
  - Static.
  - Fixed Stack-Dynamic.
  - Fixed Heap-Dynamic.

- **Dynamic shape** arrays:
  - Conformant.
  - Stack-Dynamic.
  - Heap-Dynamic.

Array Initialization

- Some languages allow initialization at the time of storage allocation:
  - C, C++, Java, C# example:

```
int list [] = {4, 5, 7, 83}
```

  - Arrays of strings in C and C++

```
char *names [] = {“Bob”, “Jake”, “Joe”];
```

  - Java initialization of String objects:

```
String[] names = {“Bob”, “Jake”, “Joe”};
```

  - Ada initialization using *arrow* operator:

```
Bunch : array (1..5) of Integer := (1 => 17,
   3 => 34, others => 0)
```

Heterogeneous Arrays

- A **heterogeneous array** is one in which the elements need not be of the same type.

- Supported by:
  - Perl: any mixture of scalar types (numbers, strings, and references).
  - JavaScript: dynamically typed language ⇒ any type.
  - Python and Ruby: references to objects of any type

Slices

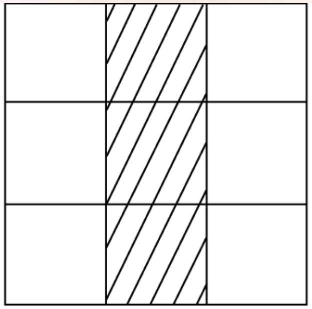
- A **slice** is some substructure of an array:
  - nothing more than a referencing mechanism.
  - only useful in languages that have array operations.

- Fortran 95 (also Perl, Python, Ruby, restricted in Ada):

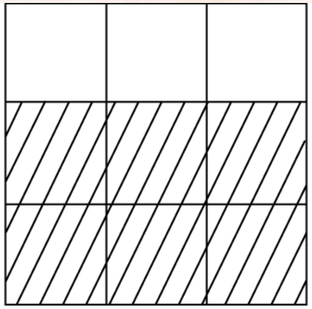
```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3, 3) :: Cube
```

`Vector (3:6)` is a four element array

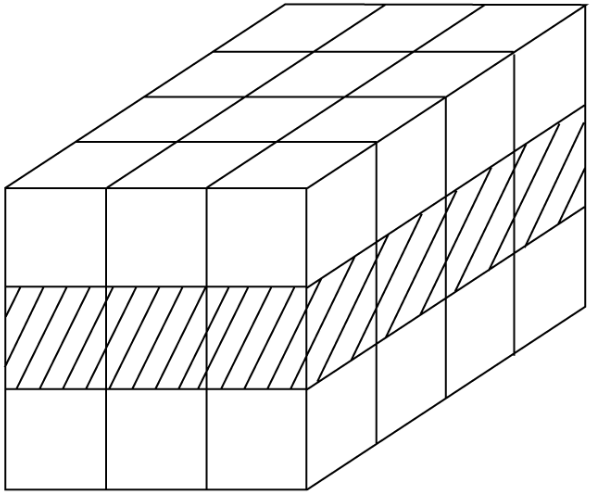
Slices Examples in Fortran 95



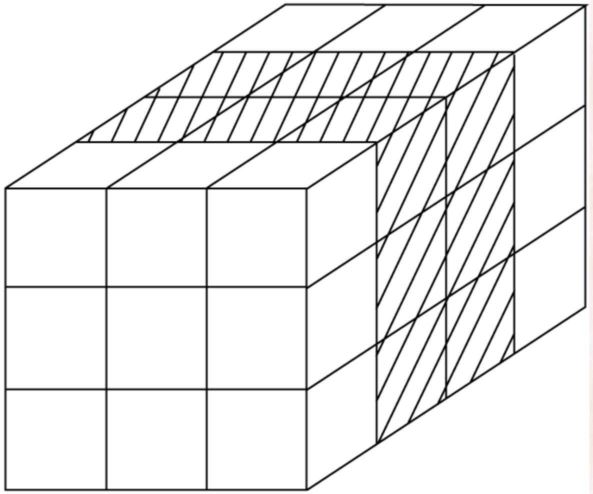
MAT (1:3, 2)



MAT (2:3, 1:3)

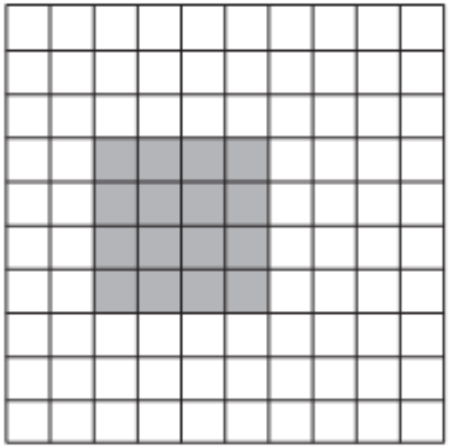


CUBE (2, 1:3, 1:4)

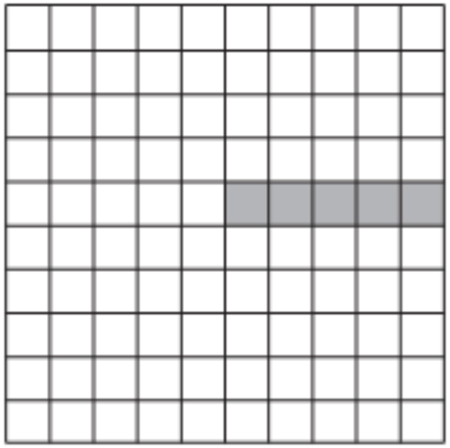


CUBE (1:3, 1:3, 2:3)

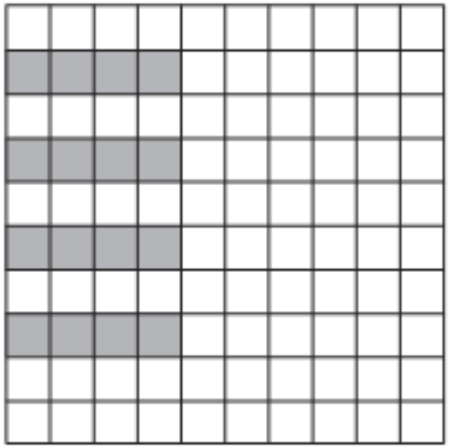
Slices Examples in Fortran 95



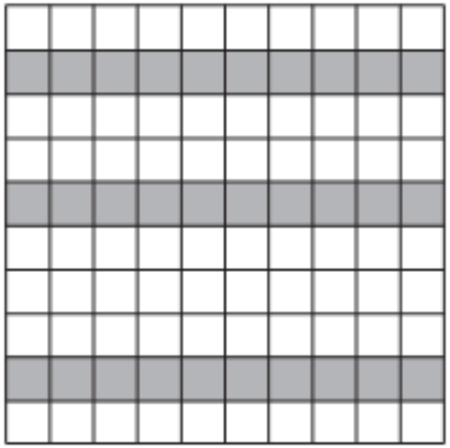
`matrix(3:6, 4:7)`



`matrix(6:, 5)`



`matrix(:4, 2:8:2)`



`matrix(:, (/2, 5, 9/))`

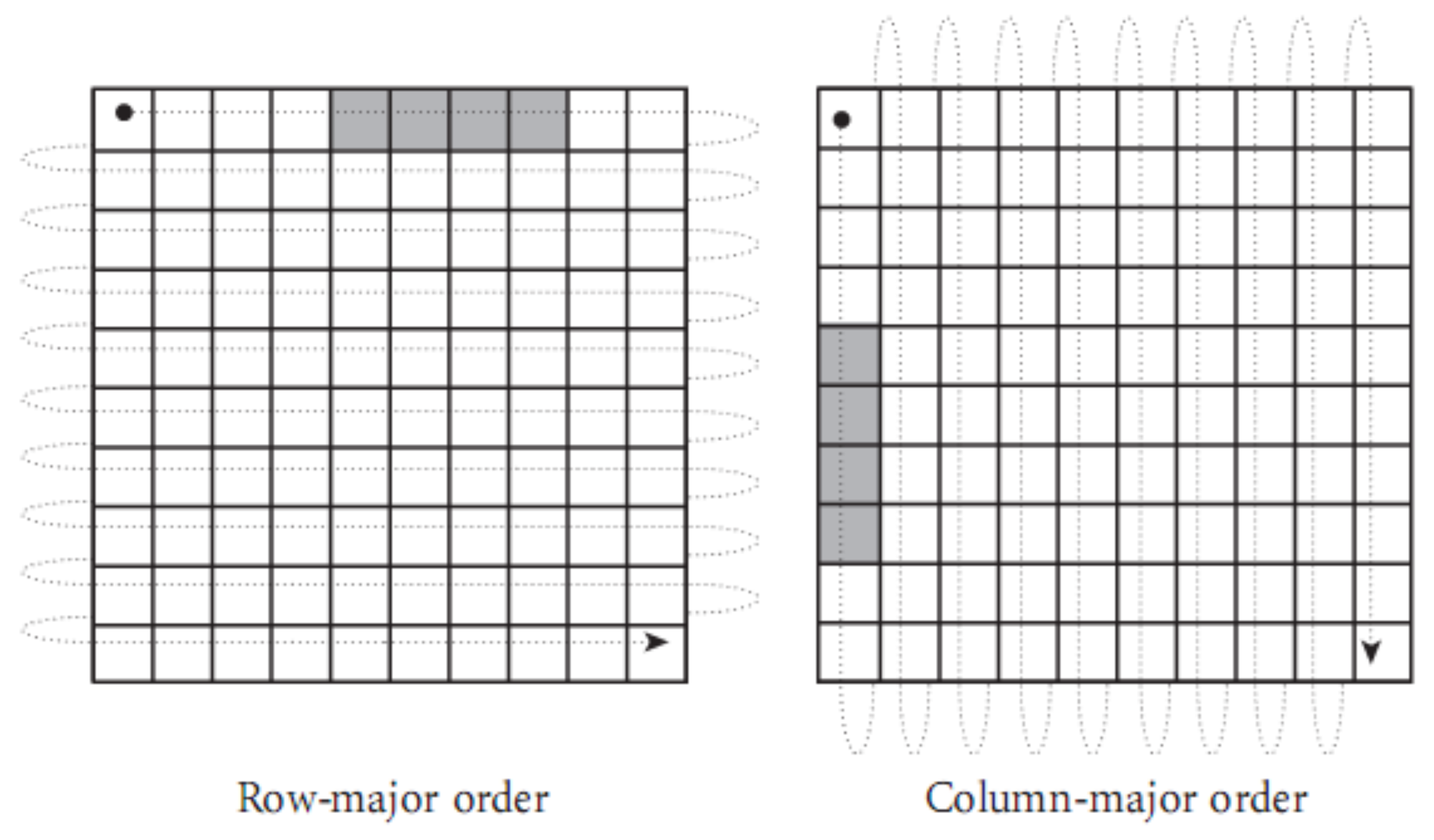
Implementation of Arrays

- Two layout strategies:
  1. **contiguous locations**.
  2. **row pointers**.

1. Contiguous locations:
   - **Column major** order (by columns) – used in Fortran.
   - **Row major** order (by rows) – used in most languages.

   - Sequential access to matrix elements will be faster if they are accessed in the order in which they are stored:
     - Why?

Row vs. Column major order



Row-major order

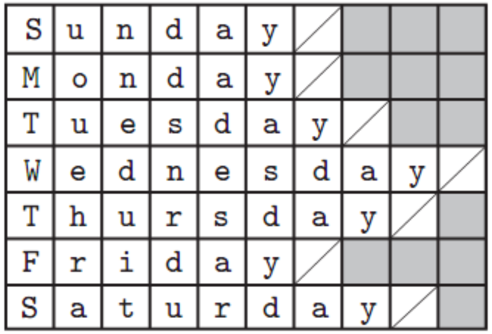
Column-major order

Implementation of Arrays

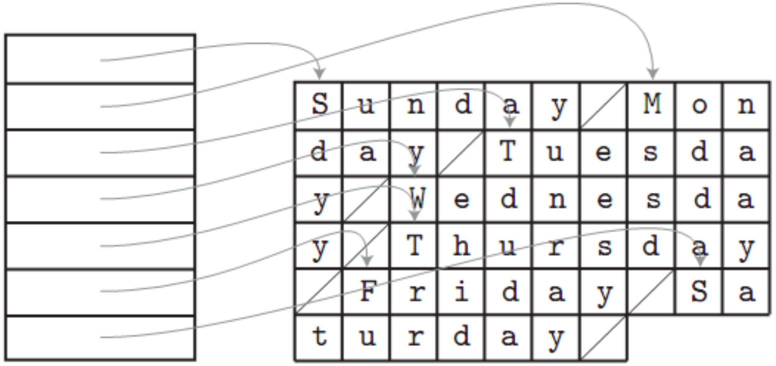
- Row Pointer layout:
  - rows can be put anywhere in memory.
  - rows can have different lengths => *jagged* arrays.
  - can create arrays from existing rows, without copying.
  - no multiplications to compute addresses => fast on CISC machines.
    - requires extra space for pointers.
  - used in Java and C:
    - C supports both contiguous and row pointer arrays.

Contiguous vs. Row Pointer layout in C

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';   /* in Tuesday */
```



```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';   /* in Tuesday */
```



Implementation of Contiguous Arrays

- **Access function** maps subscript expressions to the address of an element in the array.

- Single-Dimensional Arrays:
  - implemented as a block of adjacent memory cells.
  - access function for single-dimensioned arrays (row major):

```
A : array (L..U) of elem_type;
```

address(A[k]) =

address(A[L]) + (k – L) * element_size

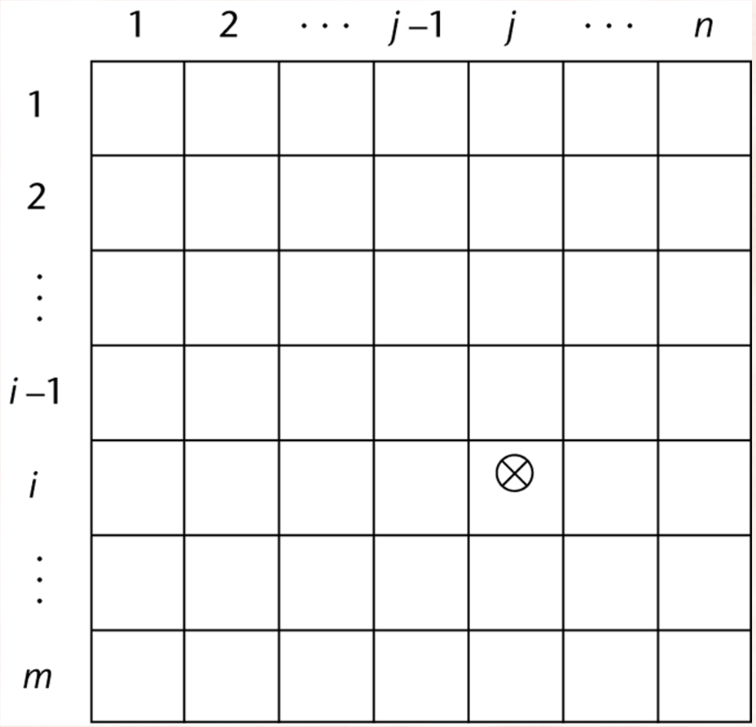
Access Function for a Multi-Dimensioned Array

```
A : array (L1..U1) of (L2..U2) of elem_type;
```

$n = U_2 - L_2 + 1$

$\text{address}(A[i,j]) =$

$$\text{address}(A[L_1,L_2]) + ((i - L_1) * n + (j - L_2)) * \text{elem\_size}$$



Implementation of Row Pointer Arrays

- Address calculation is straightforward:
  - no multiplications needed.
  - assume hardware provides an indexed addressing mode:
    - R1 = *R2[R3] (load instruction).

```
A : array (L1..U1) of (L2..U2) of elem_type;
```

Character String Types

- **Character Strings** – values are sequences of characters.
- Typical operations:
  - Assignment.
  - Comparison.
  - Concatenation.
  - Substring reference.
  - Pattern matching.

- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?

Strings in Programming Languages

- C and C++:
  - Implemented as null terminated char arrays.
  - A library of functions in `string.h` that provide string operations.
  - Many operations are inherently unsafe (ex: strcpy).
  - C++ `string` class from the standard library is safer.

- Java (C# and Ruby):
  - Primitive via the `String` class (immutable).
  - Arrays via the `StringBuilder` class (mutable, w/ subscripting).
    - `StringBuffer` for multithreading

- Fortran:
  - Primitive type.

Strings in Programming Languages

- Python:
  - Primitive type that behaves like an array of characters:
    - indexing, searching, replacement, character membership.
  - Immutable.

- Pattern Matching:
  - built-in for Perl, JavaScript, Ruby, and PHP, using regular expressions.
  - class libraries for C++, Java, Python, C#.

String Length

- **Static Length** – set when the string is created:
  - Java String, C++ STL string , Ruby String, C# .NET.
- **Limited Dynamic Length** – length can vary between 0 and a maximum set when the string is defined:
  - C/C++ null terminated strings.
- **Dynamic Length** – varying length with no maximum:
  - JavaScript and Perl (overhead of dynamic allocation/deallocation).
- Ada supports all three types:
  - String, Bounded_String, Unbounded_String.

Ada Strings

- **Static Length**:

```
X: String := Ada.Command_Line.Argument(1);
X := "Hello!";
    -- will raise an exception if X has length ≠ 6
```

- **Dynamic Length**:

```
X: Unbounded_String :=
  To_Unbounded_String(Ada.Command_Line.Argument(1))
  ;
X := To_Unbounded_String("Hello!");
```

Record Types

- A **record** is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names.
- A record type in Ada:

```
type Emp_Rec_Type is record
    First: String (1..20);
    Mid: String (1..10);
    Last: String (1..20);
    Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

Record Types

- C, C++, C#: supported with the `struct` data type.
  - In C++ structures are minor variations on classes.
  - In C# structures are related to classes, but also quite different.
    - structures are allocated on the stack (**value** types).
    - class objects are allocated on the heap (**reference** types).
  - In C++ and C# structures are also used for *encapsulation*.

- Python, Ruby: implemented as hashes.

Records vs. Arrays

- Arrays mostly used when:
  - collection of data values is homogenous.
  - values are process in the same way.
  - order is important.
- Records are used when:
  - collection of data values is heterogeneous.
  - values are not precessed in the same way.
  - unordered.
- Access to array elements is much slower than access to record fields:
  - array subscripts are dynamic.
  - record field names are static.

Unions: Free (Fortran, C/C++)

```
union  flexType {
    int i;
    double d;
    bool b;
}
union flexType ft;
ft.i = 27;
float x = ft.i; // nonsense, no type checking possible.
```

Unions: Discriminated (Algol 68, Ada)

- Include a type indicator called a **tag**, or **discriminant**.

```
type Figure (Form: Shape) is record
Filled: Boolean;
Color: Colors;
case Form is
   when Circle =>
            Diameter: Float;
   when Triangle =>
            Left_Side: Integer;
            Right_Side: Integer;
            Angle: Float;
   when Rectangle =>
            Side1: Integer;
            Side2: Integer;
end case; end record;
```

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
```

Unions: Discriminated (Algol 68, Ada)

```
Figure1 : Figure;
Figure2 : Figure(Form => Triangle);

Figure1 := (Filled => True,
            Color => Blue,
            Form => Rectangle,
            Side1 => 12,
            Side2 => 3);

if (Figure1.Diameter > 3.0) // => run-time type error.
```

Pointer Types

- A **pointer** type variable has a range of values that consists of memory addresses and a special value **nil**.
  - Provide the power of indirect addressing.
  - Provide a way to manage dynamic memory
    - a pointer can be used to access a location in the area where storage is dynamically created i.e. the heap.
    - variables that are dynamically allocated on the heap are **heap-dynamic variables**.

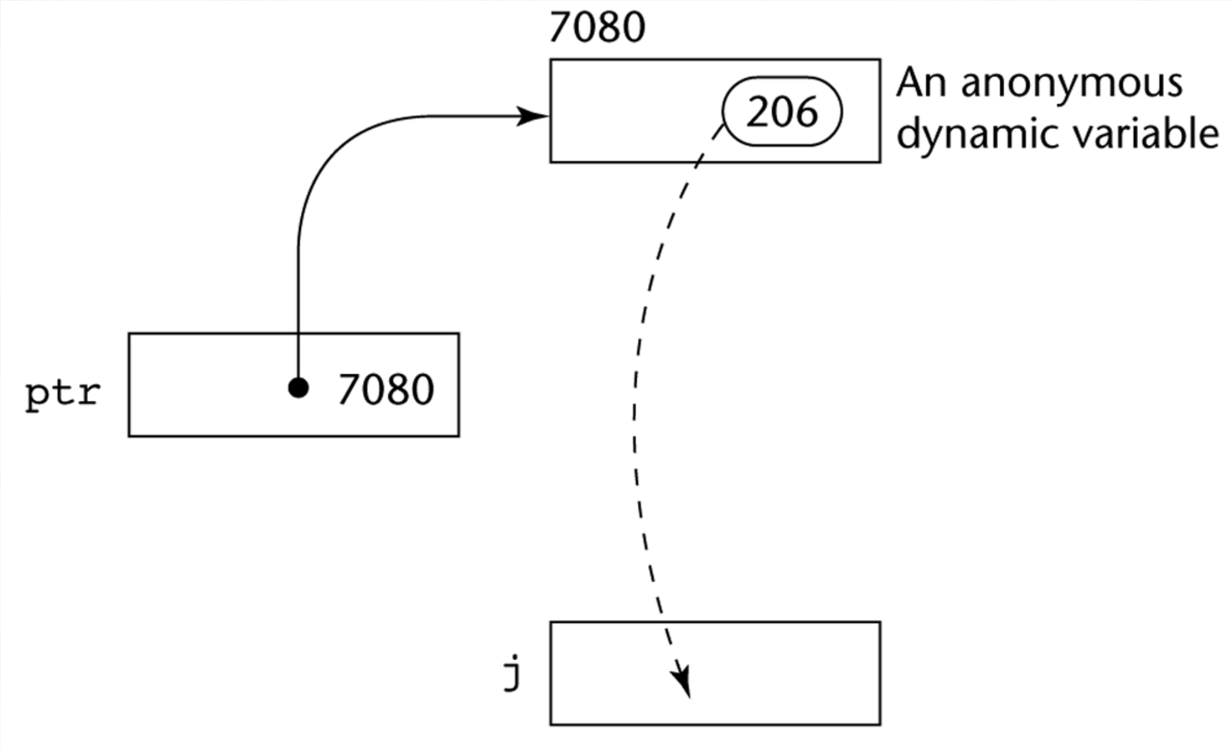
- Pointer types are defined using a type operator:
  - C/C++: `int *ptr = new int;`

Pointer Operations

- Two fundamental operations:
  - assignment.
  - dereferencing.
- Assignment is used to set a pointer variable's value to some useful address:
  - `int *ptr = &counter;` // indirect addressing.
  - `int *ptr = new int;` // heap-dynamic variable.
- Dereferencing yields the value stored at the location represented by the pointer's value
  - C++ uses an explicit operation via unary operator *:

    `j = *ptr;` // sets j to the value located at `ptr`

Pointer Dereferencing



The dereferencing operation j = *ptr;

Problems with Pointers

- Dangling pointers:
  - A pointer points to a heap-dynamic variable that has been deallocated.
  - Dangerous: the location may be assigned to other variables.
- Lost heap-dynamic variable:
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage* or *memory leak*):
    - Pointer `p1` is set to point to a newly created heap-dynamic variable
    - Pointer `p1` is later set to point to another newly created heap-dynamic variable, without deallocating the first one.

Pointers in C/C++

- Extremely flexible but must be used with care:
  - Pointers can point at any variable regardless of when or where it was allocated.
  - Used for dynamic storage management and addressing.
  - Explicit dereferencing (*) and address-of (&) operators.
  - Domain type need not be fixed:
    - `void *` can point to any type and can be type checked.
    - `void *` cannot be de-referenced.
  - Pointer arithmetic is possible.

Pointer Arithmetic in C/C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Reference Types

- C++ includes a special kind of pointer type called a **reference type** that is used primarily for formal parameters:
  - Advantages of both pass-by-reference and pass-by-value.
  - No arithmetic on references.
- Java extends C++'s reference variables and allows them to replace pointers entirely:
  - References are handles to objects, rather than being addresses.
- C# includes both the references of Java and the pointers of C++.

Evaluation of Pointers & References

- Problems due to dangling pointers and memory leaks.
- Heap management can be complex and costly.
- Pointers are analogous to `goto`'s:
  - goto’s widen the range of statements that can be executed next.
  - pointers widen the range of cells that can be accessed by a variable.
- Pointers or references are necessary for dynamic data structures, so we can't design a language without them:
  - pointers are essential for writing device drivers.
  - references in Java and C# provide some of the capabilities of pointers, without the hazards.

Type Checking

- Preliminary step: generalize the concept of operands and operators to include:
  - subprograms as operators, and parameters as operands;
  - assignments as operators, and LHS & RHS as operands.
- **Type checking** is the activity of ensuring that the operands of an operator are of *compatible types*.
- A **compatible type** is one that is either legal for the operator, or is allowed under language rules to be implicitly converted to a legal type:
  - This automatic conversion , by compiler-generated code, is called a *coercion*.

Type Checking

- A **type error** results from the application of an operator to an operand of an inappropriate type.
- **Static type checking:** if all type bindings are static, nearly all type checking can be done statically (Ada, C/C++, Java).
- **Dynamic type checking:** if type bindings are dynamic, type checking must be dynamic (Javascript, PHP).
- **Strong typing**: a programming language is strongly typed if type errors are always detected.
  - Done either at compile time or run time.
  - Advantages: allows the detection of the misuses of variables that result in type errors.

Strong Typing: Language Examples

- C and C++ less strongly typed than Pascal or Ada:
  - parameter type checking can be avoided;
  - unions are not type checked.
- Ada is strongly typed:
  - only exception: the `UNCHECKED_CONVERSION` generic function extracts the value of a variable of one type and uses it as if it were of a different type.
  - Java and C# are strongly typed in the same sense as Ada:
    - types can be explicitely cast ⇒ may get type errors at run time.
- ML is strongly typed, so are Lisp, Python and Ruby

Strong Typing & Type Coercion

- Coercion rules can weaken the strong typing considerably i.e. loss in error detection capability:
  - C++'s strong typing less effective compared to Ada's.

- Although Java has just half the assignment coercions of C++:
  - its strong typing is more effective than that of C++.
  - its strong typing is still far less effective than that of Ada.

Reading Assignment

Chapter 7 on Data Types (7.1 to 7.6)